Challenging Software Developers

Dialectic as a Foundation for Security Assurance Techniques

Charles Weir,
Security Lancaster[1]
Lancaster University, UK
c.weir1@lancaster.ac.uk

Awais Rashid
Bristol Cyber Security Group
University of Bristol, UK
awais.rashid@bristol.ac.uk

James Noble
School of Engineering and Computer Science
Victoria University
Wellington, NZ
kjx@ecs.vuw.ac.nz

Abstract

Development teams are increasingly expected to deliver secure code, but how can they best achieve this? Traditional security practice, which emphasises ‘telling developers what to do’ using checklists, processes and errors to avoid, has proved difficult to introduce. From analysis of industry interviews with a dozen experts in app development security, we find that secure development requires ‘dialectic’: a challenging dialog between the developers and a range of counterparties, continued throughout the development cycle. Analysing a further survey of sixteen industry developer security advocates, we identify the six assurance techniques that are most effective at achieving this dialectic in existing development teams, and conclude that the introduction of these techniques is best driven by the developers themselves. Concentrating on these six assurance techniques, and the dialectical interactions they involve, has the potential to increase the security of development activities and thus improve software security for everyone.

1 Introduction

Software security and privacy are becoming major issues: almost every week we hear that yet another organisation's software systems have been compromised [18]. While there are many aspects to an organisation's security and privacy, the software used clearly has a significant impact on whether such breaches happen. Therefore, the effectiveness of developers at creating secure software is vital[2].

Unfortunately, there is evidence that the developers are not delivering this security. A recent report from Veracode [50] concluded that "more than 85 percent of all applications have at least one vulnerability in them; more than 13 percent of applications have at least one very high severity flaw". A report from Microsoft [34] analysed storage and collaboration Software as a Service applications, and found a wide range of issues, including 28% of storage apps not supporting data encryption. The Ponemon Institute carried out a IBM-funded survey in 2015 of 640 individuals from organizations developing mobile apps in the US [42], and found that 77% believed that securing mobile apps was 'very hard', and that 73% believed that developer lack of understanding of security issues was a major contributor to the problem.

All these surveys suggest that many errors were avoidable; developers could have made choices that would have prevented the issues. These studies demonstrate that existing industry practices are insufficient to provide the application security and privacy we need.

To address this problem, we could have looked for improvements to environments and APIs; we could have looked at tools to automate security improvements; or we could look at ways in which we can help programmers themselves to improve security given existing constraints. All are valuable approaches; we chose the third option. In this article, therefore, we investigate programmers' security practices, using expert surveys to find out what works in practice.

1.1 First Phase: App Security Expert Interviews

Faced with a broad range of diverse kinds of developer, we initially limited our scope to mobile app development. The initial research question we formulated during the work was therefore:

What techniques and ideas lead to the development of better secure app software?

To answer this question, we needed an understanding of what works in the real world. We therefore conducted a Constructivist Grounded Theory [12] study, involving face-to-face interviews with a dozen experts whose cumulative experience totalled well over 100 years of secure app development. Our early analysis of the interviews [54,55] found a wide range of difference between interviewees, and concluded that there was little consensus in the industry.

Further analysis identified that the most important and successful secure development techniques share a quality we call 'dialectic,' meaning learning by challenging. Dialectic techniques use dialog with a range of counterparties to achieve security in an effective and economical way. The increase in security comes from the developers' continued interaction with the resulting challenges, not from passive learning. This suggests that we need developers to actively seek out arguments and challenges in order to improve software security.

[2] Throughout this paper we use 'secure' and 'security' to refer to the security and privacy aspects of software development; and 'developers' to refer to all those involved with creating software: programmers, analysts, designers, testers, and managers.

1.2 Second Phase: Intervention Expert Interviews

Given the suggestion that dialectic techniques increase security, we wanted to find ways to introduce such techniques to development teams who were not yet using them. We observed that none of the techniques, nor the dialectic concept, appeared specific to mobile app development. Indeed, though we had chosen the first survey's participants for their experience with app security, in discussing the questions most had cited experience with a far wider range of software domains. We therefore conducted a second survey to find ways to induce teams to adopt dialectic techniques, extending our scope to cover professional software development in general.

Our research question for this next step, therefore, was:

What interventions can change the environment for members of a development team to achieve good security?

In addressing this question, we considered motivational factors, choice of tools, supporting processes and potential blockers, culture, awareness, training and skills. We observed that previous research in software security has concentrated on the negative: identifying vulnerabilities and making checklists of issues to avoid (see Section 2.2). Yet developers are motivated by positive aspects of their work such as building and helping others [9], so we chose instead to concentrate on positive approaches to improving security.

Using the same Grounded Theory approach as the first phase, we accordingly interviewed sixteen specialists in developer software security, asking them about these topics. These specialists ranged from senior experts in the major multinational online service providers to solo consultants.

From our interviews, we identified a consistent perception of the developer as an active agent in their own right, whose decisions could be influenced by security experts but not controlled by them. More specifically, we identified six assurance techniques preferred by the experts, and a range of counterparties with whom the dialectic takes part. We analyse these in detail in this paper, highlighting the dialectic effect on security.

1.3 Contribution and Overview

The contribution of this work is a new perspective upon the introduction and improvement of security in software development teams. This paper provides:

- A theory, 'Dialectical Security,' to provide a basis for analysis of developer security techniques;
- Identification of six assurance techniques preferred by industry experts;
- Analysis of each of these assurance techniques in terms of its dialectic counterparties; and
- Examples and approaches for introducing the techniques.

In the rest of this paper, Section 2 explores existing work; Section 3 explains the methodology in more detail and describes the research participants; Section 4 describes the results of the surveys, introducing dialectic and analysing how it applies in each of six assurance techniques; Section 5 discusses the research as a whole; Section 6 explores threats to validity; and Section 7 provides a conclusion and suggests approaches for future work.

2 Background and Related Work

We looked for related work in three areas:

- How and why programmers learn security,
- Resources to help programmers improve security, and
- Techniques to help teams improve

2.1 How Programmers Learn Security

There is relatively little literature on how programmers learn, whether about novices or professional programmers. Johnson and Senges [27] studied how programmers learned to function in a complicated organisation, Google. They concluded that the majority of programmer learning there was peer learning, facilitated by strong corporate standards and culture; obviously the results are limited to that organisation. Enes and Conradi [17] used interviews to discover how professionals, including a few programmers, acquired their expert knowledge. It concludes that the preferred learning mechanisms are all informal ones: especially on-the job training and personal interaction. Murphy-Hill et al. explored how developers find new software tools [35]; the research was more wide-ranging and included both an initial survey and a larger scale, 79 participant, diary-based survey. They concluded the event is relatively rare, and happens more through joint working than recommendations

Balebako et al. surveyed and interviewed over 200 app developers, and concluded that most approached security issues using web search, or by consulting peers [6]. A survey by Acar et al. reached similar conclusions; and they also determined experimentally the surprising result that programmers using digital books achieved better security than those using web search [1]. Yskout et al. tested experimentally the effect of using security patterns in server design; the results suggested a benefit but were statistically inconclusive [61].

Xie et al. [59], interviewed programmers to investigate why they believed they made security errors; they found a consistent tendency to treat security as 'someone else's problem'. Though the results are limited in scope to highly experienced developers in US companies, it seems reasonable to conclude that the conclusions may apply more widely.

Votipka et al. [51] compared the approaches of professional penetration testers with those of developer testers, concluding that their approaches are similar; the experience of the pen testers leads to better security findings; but the closeness of developer testers to programmers can lead to better outcomes. They therefore recommended that penetration testers work directly with programmers.

2.2 Resources to Help Programmers

Turning to resources, one might expect that an effective contribution to app security would be books explaining how to do secure development. The recently-published book, Bell et al.'s Agile Application Security [10] does this, explaining security techniques to application development, and agile and programming concepts to security experts, and providing a good deal of practical information, though it is not yet widely read[3].

Few other resources seem to be helpful to developers at a higher level than code. There are many good works describing the theory and practice of software security, such as Gollman's 'Computer Security' [21], Schneier's 'Secrets and Lies' [44] and Anderson's 'Security Engineering' [3]; these are particularly valuable for introducing the concepts of 'whole system security', but all work at a level that isn't helpful as anything but background reference for a software developer.

Instead, the most useful learning books for software developers tend to be those that convey information in a relatively terse and readable form, and in manageable chunks. A good example is Howard et al.'s '24 Deadly Sins of Software Security' [24]. An online version is the community-written OWASP Top Ten Risks sites [39], a widely accessed[4] resource detailing specific programming issues and how to avoid them, which has versions for a variety of platforms. Their authority and availability make them effective, though they consider few 'whole system security' issues. Acar et al. [2] surveyed a range of web-based software

[3] Based on Amazon.com rankings as at January 2018

[4] Based on Google rankings in February 2016

security resources available for developers, concluding that there are distinct gaps in the coverage, and that not much of it was suitable for developer learning.

Other books and sites are targeted specifically at particular platforms. Book examples include 'Android Security Internals', 'Learning iOS Security' [7]; or Core Security Patterns for J2EE [46]; web resources include Apple's for iOS [4], Google's for Android [22] and Microsoft's [32].

Some of the most popular[5] security books are the ones with a platform-specific 'Black Hat' (attacker) approach. For example the Android Hacker's Handbook [16], and its corresponding versions for iOS and web apps, contain a good deal about exploits against the operating system, a certain amount about analysing existing software, but little about how to guard against exploits as a developer.

Most programmers tend to use web search and discussion sites such as Stack Overflow as their primary source of information on security [1]. Unfortunately these lack overview discussions [8], making them valuable in helping programmers sort out problems they know they have, but does not point out problems that they do not know they may have; most security problems are likely to be of this second type.

Discussion sites have a second problem related to security: their answers on security matters tend to be of questionable accuracy especially when they quote code. Acar et al. [1] analysed answers on Stack Overflow to app security questions, worryingly finding around 50% of solutions to a set of security questions to contain insecure code snippets.

2.3 Techniques to Help Teams Improve

One approach to helping teams with security is providing code analysis tools: seven research groups [15,28,36,37,43,45,58] have created security defect detection tools to help developers improve code, using feedback via IDEs or elsewhere. Xie et al. [60] explored the impact of their IDE-based security analysis tool for web applications on a sample of 21 students and 6 professionals and found:

Even when creating secure code is relatively easy … users still need to be motivated to make the needed changes.

One might expect the most effective approach to be a prescriptive set of instructions telling programmers what to do, a 'Secure Software Development Lifecycle' (SSDL) such as those promoted by Microsoft [33] and others. However Conradi and Dyba [13], among others, identified that programmers have difficulty with, and resist learning from, formal written routines, and this appears to have been the experience with SSDLs in practice [19]. So, since about 2010 several of the SSDLs have been replaced by 'Security Capability Maturity Models' [31,40], to allow management influence on software security at a corporate level based on measurements using checklists of processes used to improve security. These are effective [30] at defining what development teams should achieve, but provide little help to developers on how best to achieve it.

Some research teams have explored the impact of training and external involvement on teams' delivery of secure software. Türpe et al. [49] explored ethnographically the effect of a penetration testing session and workshop on 37 members of a large geographically-dispersed project, concluding that it failed mainly because the workshop consultant highlighted problems without offering solutions. Poller et al. [41] followed an attempt to improve long term security practices in an agile development team of about 15 people, which failed because pressure to add functionality meant that attention was not given to security issues., and because their normal work procedures did not support goals such as security improvement. The authors concluded that successful interventions would need "to investigate the potential business value of security, thus making it a more tangible development goal;" and that security is best promoted as a team, not individual, effort.

[5] Based on Amazon.com rankings as at January 2016

Work by Ashenden and Lawrence [5] took a different approach. They used an Action Research method to investigate and improve the relationships between security professionals and software developers in a single company, and found the approach effective in improving communication. Lopez et al. [29] used discussion workshops based on ethnographic studies to encourage similar communication. No evidence is yet available of longer-term impact from either intervention.

Another approach is to investigate the economics of different software assurance techniques. Such et al. [48] surveyed 150 security specialists to analyse the economics of different software assurance techniques related to security. The survey asked about twenty pre-defined assurance techniques, finding Public Review and (tool-based) Static Analysis to be the most cost-effective, and Formal Verification and Cryptographic Validation to be the least. Interestingly the researchers also identified frequently-used combinations of techniques, finding that the combination Architectural Review, Configuration Review, (manual) Penetration Test and (automated, web-based) Vulnerability Scan was seen as most cost-effective.

2.4 Limitations of Existing Literature

Consistent in all this literature, is a lack of knowledge and theory as to how successful industry development teams achieve software security. This article addresses that lack.

3 Methodology

This section explains our choice of methodology, Grounded Theory, and introduces the two sets of research participants.

3.1 Choice of Methodology

Since there was little existing theory on developers and app security, we had no basis for an experimental approach; although we had a target – improving developer security – we had no theory to test. Written or email surveys could be useful to find out current practice from a list of options but are unsuitable for the kinds of open question that generates theory.

Our purpose in the research was to generate knowledge about good approaches to achieve secure development. Two perceptions drove the choice of research approach:

- We had found few resources discussing how to tackle app development security.
- Existing literature tended to be negative in approach, listing things the developer must not do; this contrasts with the kinds of books preferred by developers, which we had observed to be positive in outlook.

Since our aim was to generate a novel theory, we chose Grounded Theory as our primary research method. Our studies used semi-structured interviews with industry experts. We chose interviewees opportunistically; our connections in industry provided introductions to a number of successful, and mostly senior, practitioners with considerable experience of helping teams achieve software security.

The first ‘App Security Expert’ phase of interviews took six months, and involved 12 such experts; the second, ‘Intervention Expert’ phase took a similar period, and involved 15 interviews with 16 experts. The Appendix contains the protocols used for each set of interviews.

Both studies were separately approved by the Lancaster University Faculty of Science and Technology Research Ethics committee.

3.2 Grounded Theory

Grounded Theory (GT) [20] uses textual analysis of unstructured text to make theory generation into a dependable process. It requires line-by-line analysis of everything relevant that is available to the researcher: interview transcripts, relevant research literature, field notes from observation and anything else that can reduce to text form. The process is iterative, with analysis of initial findings from interviews or similar leading to changes in the research thrust and direction.

We used the Constructivist GT variant [12], acknowledging the effects of the researcher on the results. We followed the principles for software engineering GT described by Stol et al. [47]. The interviews consulted the participants as experts rather than as subjects, addressing what each had found to be most successful in their experience in secure software development. To encourage positivity, we used elements of Appreciative Inquiry [14] in our questioning: the 'Discovery' of best practice and the 'Dream' of ideal practice. Questions avoided discussion on what does not work and concentrated on the most effective practice known to each interviewee.

We recorded interviews and transcribed them manually; organisation of the data used the commercial tool NVivo; coding, memo creation and sorting were all by the lead author.

The Grounded Theory analysis involved scanning each text line-by-line, highlighting points of interest. One or more codes are then assigned to each point of interest, with new codes created as required to identify the ideas and concepts involved. Codes were reused across documents, so that a given code represents the same idea throughout the research. During that process, the coder also 'memoed' ideas and thoughts about how the terms may be interrelated. Gradually, as codes were assigned, the coder merged codes with similar meaning, and organised the resulting codes into groups called 'categories'.

The coding process for both surveys generated a total of 2,140 coding entries to 340 distinct codes, grouped into 12 high-level categories.

The Grounded Theory methodology derives from social science, and in contrast to many computer science methodologies, dual coding is not a GT practice [12]. Therefore, inter-rater agreement calculations are not applicable. Instead researcher interaction takes the form of discussion of the codes, findings and outcomes, and in this case was based around the writing of papers.

In GT analysis, the aim is to find a single overarching Core Category: the category which covers the most interesting features in the data. Data gathering is considered complete when further data received does not lead to significant new concepts, a situation termed 'theoretical saturation'. The Core Category then forms a basis from which to construct theory. Section 4 presents the theory of the core category we found: Dialectical Security.

Finally, we sent a detailed report describing the findings from each survey to each of the interviewees in that survey.

3.3 Research Participants

Interviewees were recruited opportunistically; connections in industry and academia provided introductions to successful, and mostly senior, practitioners with considerable experience of helping teams achieve software security.

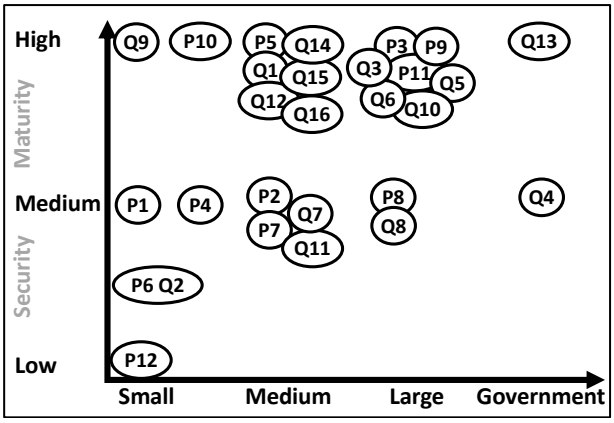

Figure 1: Organisation Size and Security Capability

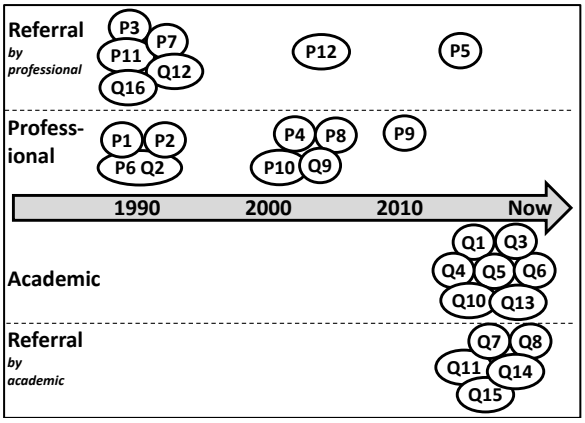

Figure 2: Recruitment Method

Figure 1 introduces the participants: those in the first 'App Expert' survey have IDs starting P; those in the second 'Intervention Expert' survey have IDs starting with Q. The horizontal axis indicates their organisation size: Small for consultants and up to 10 people; Medium up to 1000 people, else Large or Government. The vertical axis provides a subjective estimate of the 'secure software capability maturity' [26] of the software teams they work with, as follows:

Low — Little or no awareness or activity related to software security

Medium — Aware of and addressing security issues, typically including some developers with good security knowledge.

High — Experts at software security, within an organisational culture that assigns it a high priority.

To show further the range of participants involved, Figure 2 summarises the recruitment process. We approached various people; the horizontal axis indicates how long we had known the people approached. Some we interviewed directly, and these are shown closest to the axis. Professional and personal contacts are shown above the axis, and contacts from academia—mostly encountered at industry-academic workshops—below. In other cases, the contacts we approached referred us to others, and the resulting interviewees are shown further from the axis.

Most of the participants were based in the UK other than P11, Q5 based in the USA; Q7 in Sweden; and Q10, Q13 in Germany. All UK interviews were face-to-face; others used video conferencing.

Only one person was interviewed in both surveys (P6 Q2, shown together); the requirement in the second survey for a wider remit than app development discouraged other repeat interviews. Each participant was in a different organisation, other than P3/P11, Q14/Q15 and Q12/Q16 – the latter two who were interviewed together. Table 1 in Section 4.2 has the participant job roles and the company descriptions.

We chose experts in secure *software development*; many were therefore predominantly developers first and security experts second. As Table 1 shows, only fourteen of the twenty-seven had job titles or company missions specifically related to security. Their qualification as experts was based on their reputations, either directly from the researchers' knowledge; or as validated by the people who referred them to us.

In the following sections, quotations from the interviewees are in *italics*. We have edited them to protect confidentiality and indicate context: square brackets show additions and replacements; ellipses show removals.

4 Analysis

4.1 Introducing Dialectic

The Core Category that emerged from the Grounded Theory analysis of the first 'App Security' phase of interviews was the nature of the developer's interaction with external parties: a friendly adversarial interaction. The word 'dialectic' had surfaced in our earlier work [54], and we realised it was the common theme to all these interactions. 'Dialectic' is the finding out of knowledge, especially logical inconsistencies, through one person questioning another[6].

I think part of the good practice guide is embedding questioning. How, why, what, where, when. It is almost asking developers to ask the questions that begin with those words and thinking of a set of questions that begins with those words. What if we did that? How would we do that? When could we do this? (P5)

If there is something in the application that is a particularly sensitive area… you probably want a security guy to come and sit for a few days and have a look at it: someone who can work with the develop team but who hasn't been part of that development team, in a way. So, somebody who can ask the stupid question - I think that is really important, to bring in an external person. (Q11)

Dialectic surfaced as the Grounded Theory 'Core Category' with the observation that many of the techniques used for security depended on someone or something 'surprising' and 'questioning' developers:

There is a separate Security Review system for, so if you are doing code that impacts security in your judgement, it goes to people who are security experts who will do the security review and they find stuff. (P3)

Every production change has to go through the Change Advisory Board, … [with] a really good cross section… of developers, system administrators, database administrators, QA, business security …, and everything gets a fair, and good critique. (Q16)

If you open [a security code review tool, you might see] "Now you are 8 out of 10, okay. Nice work, but oh, but you have failed the security mis-configuration." It explains what security misconfiguration is, … it gives me external resources, … and a sharing knowledge base, where you get can more information on what to fix, and how to remediate it. (Q7)

Normally the way the [Penetration Testing] report would be dealt with is we would sit down in a sprint planning meeting and we would go through the report: us, the client, sometimes the tester… and we would discuss the findings and produce work that would be added to the next sprint to address what had been found. (Q1)

Dialectic first appeared as the technique used by the Greek philosopher Socrates [57]. Dialectic addresses a particular weakness in much existing secure development research, namely that emphasis on code level security often misses possible exploits based on the functionality and system design – or may prove unnecessarily costly:

So implicit in [conventional thinking] is the notion that programmers decide what they are doing [only] in code … being told to put something in place without them understanding the greater implication. (P9)

[6] Although the 'Dialectic' concept came from the first phase analysis, the interviews in the second phase confirmed it, and some of the quotations in this section come from those interviews.

Instead, developers need to think about the approaches an attacker might use to gain benefit from the system they are producing, and then to decide how to thwart those approaches.

Yes, the question is 'who is the attacker, who is the bad guy, who is the threat model you are dealing with?' (P3)

This is a significantly different approach from 'normal programming', as it requires developers to think in unfamiliar ways.

They are very devious. There are exploits that they have realised which are, well, you wouldn't really think like that if you were an engineer (P2)

It was not difficult to work out why our experts should view dialectic as a solution. Unlike other forms of software quality such as performance or reliability, security involves the idea of an attacker who will use remarkably different thinking. To deal with such threats, a developer needs to think 'outside the box;' the easiest way to achieve that is with challenges from others.

The dialectic continues throughout the development cycle; and in each case, it is always two-way: the experts state that the increase in security comes from active interaction with the challenges, not from a passive understanding of the challenge.

Our theory does not suggest necessarily that dialectic provides the *best* techniques for achieving security; it does suggest, however, that dialectic *is associated with* effective software security.

4.2 Assurance Techniques

From the second 'Intervention Expert' phase of interviews, we derived further theory identifying six software assurance techniques as the most effective[7]. These exclude training and motivation activities, and are as follows:

Threat Assessment	Identifying and ranking the threats to computer software, a component, or an IT system.
Stakeholder Negotiation	Discussion and negotiation with stakeholders, such as product managers, on security choices
Configuration Review	A review of the way a system or its software has been configured to see if this leads to known vulnerabilities, using manual checking software versions or automated build review scanners.
Vulnerability Scan	The process of using an automated scanner on a web application or network to identify vulnerabilities.
Source Code Review	The manual examination of source code to discover faults that were introduced during the software development process.
Penetration Test	A simulated attack on a component or system, carried out by a security expert using similar techniques to that of a real- world malicious attacker.

For consistency with other literature, the definitions above are taken from Such et al. [48], except for Stakeholder Negotiation, which was not discussed in that paper. Note that the coder was not aware of that paper at the time of coding, and the assurance technique names derived from the Grounded Theory analysis were different.

[7] The 'most effective' criterion came from the second phase interviews; both sets of interviewees discussed assurance techniques and some of the quotations in this section come from the first phase.

Table 1: Interviewee Discussion of Assurance Techniques

ID	Organisation Type	Main Role	Vuln-erability Scan	Source Code Review	Stake-holder Neg.	Penet-ration testing	Threat Assess-ment	Config-uration Review
P1	*Bespoke app developer*	*Developing apps for business clients*						
P2	*Mobile phone manufacturer*	*Leading large security team*					2	2
P3	*Operating system supplier*	*Developing user-facing web services*		4				2
P4	*Smart card specialists*	*Developing smart card software*						
P5	*Security-related SAAS supplier*	*Promoting a secure service*		2			1	1
*P6**	*Security consultant*	*Consultant*					1	1
P7	*Mobile phone manufacturer*	*Architecting OS services*		2			2	
P8	*Telecoms service provider*	*Architecting web services*				1	1	5
P9	*Bank*	*Protecting web-based services*			1		8	1
P10	*Secure app tech. provider*	*Architecting and app technologies*			1			3
P11	*Operating system supplier*	*OS security enhancements*					9	7
P12	*Bespoke app developer*	*Developing apps for business clients*					2	3
Q1	*Outsourced software developer*	*CEO*	5	9	2	18		13
*Q2**	*Security consultant*	*Consultant*	5				1	
Q3	*Security and military supplier*	*Team leader*	10	9	3	2		
Q4	*Research organisation*	*Research and support*	3			2	1	
Q5	*Operating System Supplier*	*Security team leader*	5	3	4	8		2
Q6	*Security and military supplier*	*Security expert*	1	4	8	1		
Q7	*Software security tool supplier*	*CEO*	19	1				
Q8	*Telecommunications provider*	*Security expert*	7	2	1	1	1	
Q9	*Security consultant*	*Consultant*	5	6	8	4	1	
Q10	*Software package supplier*	*Security expert*	4	1			5	2
Q11	*Software security service supplier*	*Training and consultancy*	10	4	3	8	1	
Q12	*Telecoms service provider*	*Security expert*			16	3	3	
Q13	*Research organisation*	*Research and consultant*	11	2	1		5	
Q14	*Outsourced software developer*	*Principal engineer*			7			
Q15	*Outsourced software developer*	*Security expert*	16	6			1	
Q16	*Telecoms service provider*	*Team lead*	4	9	1			

4.3 Participant Discussion of Assurance Techniques

Table 1 shows the participants' discussion of these techniques in both phases. The numbers indicate the percentage of participant words coded to each assurance technique: the share of the discussion devoted to that technique within that interview. Cells are highlighted based on their values. The most discussed techniques are on the left; less discussed ones towards the right[8].

The table shows the substantial difference in results we saw according to the questions asked of experts. Some of the difference between the groups can be attributed to the difference in coding: in the second phase we were explicitly coding for 'interventions,' which include the assessment techniques. However, we observe a surprising lack of reference to forms of Vulnerability Scan and Stakeholder Negotiation in the first survey.

4.4 Dialectic Counterparties

The theory also identifies specific counterparty roles for this dialectic interaction:

- Members of the developer's team
- Stakeholders, such as product managers
- Automated code analysis tools
- Security experts
- Penetration test experts
- Other development teams
- End users and operations staff.

Note that these interactions are not limited to human forms of questioning; tools for automated code analysis also provide dialectic challenge.

Sections 4.5 to 4.10 explore in detail how dialectic applies in each assurance technique, discussing the counterparties for each. Each section discusses the context of the problem, and outlines how the technique solves it, exploring how the interviewees use the technique, backing the discussion with quotations from interview participants

4.5 Threat Assessment

Threat Assessment is the activity of identifying and ranking the threats to computer software, a software component, or an IT system as a whole [48].

4.5.1 Context

Any system can be broken with sufficient determination, ingenuity, and resources.

Every security system can be broken. Period. There are even ways of getting the certificates off a phone, by freezing the phone and reading the memory. There is nothing you can do to stop a truly determined person to getting in, short of dropping it into a nuclear furnace. The best you can do is make it difficult enough for them, that they will lose interest – that it's not worth the trouble. (P7)

I quickly realised that no system is ever unbreakable (P9)

As a result, secure development is not a matter of making a completely secure system. Instead, it becomes a question of which defences to implement: where one should spend the time and effort defending the

[8] As previously discussed, the asterisked P6 and Q2 were different interviews of the same expert.

system to deter the largest and most damaging potential exploits. Making those choices requires an understanding of the potential attackers:

I think it is actually very important to understand the motivations behind why somebody is hacking the system. We try to address the motivations of the attackers, versus the technical aspects - just locking it down for the sake of locking it down. (P11)

There are clear reasons why someone would want to attack a bank, but actually the real reasons for attacking a bank are very seldom to do with trying to get financial rewards. It is much more around what information you can get about people. Banks hold information about people. So [it might be] a private investigator who is trying to track someone, or a hostage situation, where people might have done things, or simply learning more about behaviour. (P9)

4.5.2 Solution

To address this, security experts use 'Threat Modelling' techniques: identifying the causes or motives and possible scenarios for a full range of threats to the systems in question.

Your answer to any kind of security question anywhere should almost always start with a threat model. (Q9)

I think the things that are the most challenging around security really are trying to understand the threat landscape and trying to understand how threats are realised. (P2).

Several interviewees indicated that developers must drive the threat modelling process.

You need developers to do threat models, but you need developers to understand how the attackers brain works, what is the methodology, the way I know how to do that is to make them do the steps, make them ask the questions. (Q11)

4.5.3 Execution and Counterparties

The counterparties discussed for Threat Assessment included other developers:

Threat modelling: what I see as the big benefit here is, is that it is not that you are coming up with a list of issues that product of the team has, it is more putting the team into the perspective, to think about the functionality from a different aspect, from a different point of view, that that brings up a list of issues, no doubt about that, but it also fosters the understanding to see the coding from a different side, to also to think about it when you create code: what else can there be? (Q10)

Some experts emphasized the importance of including more senior stakeholders:

If [the developers] don't know what a threat model is, then tell them what a threat model is, and the simplest explanation in the world is 'who is going to attack you, and where is the gold?' 'Where's the gold' answers need to come from managers, and company owners. Not from developers because their answers will be totally wrong. And actually that is a big learning that you can give to any company when you go in. Whatever the CEO's threat model is, explain it to all of the devs and go 'if you don't know what the worst nightmare is for your CEO, then how can you be expected not to make that cock up' (Q9)

And the Security Experts themselves:

I was involved in a lot of conversations about trying to think about doing really evil things, so I think in order to protect people from harm we have to think about how harm can be done. So, brain-storming bad intent is part of the life, really. (P5)

And Penetration Test Experts:

One of the things I like to do with the [penetration testing] guys is if you sit down and say, "what are all the different ways you could subvert this system." It is quite common to come up with 20, 30, 40, 50 in five or ten minutes of brainstorming. I bet you, you wouldn't think of half of them. (P2)

Particularly with development teams using agile approaches, this Threat Assessment process continues informally throughout the initial development project, and into the subsequent deployment and later lifetime of the product. The most security-capable teams incorporated attacks and motivations found throughout the software product lifecycle.

The other thing is … [to] reward proactive thinking and this is two levels of that: trying to think what could happen next, how could it go wrong, what am I missing, but then the next level of reward, is rewarding people for research. And thinking about how to do harm. Actively encourage them to think like a hacker. (P5)

4.6 Stakeholder Negotiation

Stakeholder Negotiation is the activity of discussion and negotiation with stakeholders, such as product managers, on security choices.

4.6.1 Context

Merely identifying the possible attackers and exploits does not itself deliver software security. The need is to prevent them from causing damage to users, stakeholders or others.

For businesses it is a risk-based approach which they need to understand and neither [management nor programmers] should be caring about actual nitty gritty details of coding which is just an artefact of the whole thing. (P6)

Given the Threat Assessment, a development team can take the list of possible attacks, and work out possible mitigations for each. These mitigations will each have costs in development time, commitment, finance, and sometimes usability. The team can estimate financial and other costs for each. How, though, do they make the decision which to implement?

Similarly, there is a major challenge for products after the end of the initial development project:

Like many things that get delivered in a project, the project ends, and interest dies with it. Unfortunately. And I think you lead into a significant challenge in securing things on an operational basis. (P8)

The decision of what aspects of security to implement is a commercial one. Implied in every decision about software security is a trade-off of the cost of the security against the benefit received. Every security enhancement needs to be weighed against other uses of the investment—financial, time, usability—required.

4.6.2 Solution

The solution is for development teams and security experts to express risk and costs to stakeholders (project managers, senior management, customers) in terms they can understand and use to question about security concerns against other organisation and project needs.

[Costly development approaches aren't] suitable for a lot of start-ups. And the same goes for security. You're going to have to make a security decision upfront… [When I started] a project I'd go back and ask [my customer] … "You do realise this [information] can be seen." It goes from there: 'how secure do you want it to be?' You have to show that there's a problem first I think" (P1)

4.6.3 Execution and Counterparties

The primary stakeholders are those who prioritise development tasks, such as product managers. Evaluating and expressing the threat in business terms requires discussion and experience, so other Team Members and Security Experts are also important:

The teams take a risk management approach, full stop. For every risk, we try and weight it on two axis – one is the severity, if it happens. So, ignore the likelihood, how bad is it if it happens? And the other is the likelihood, how likely is it to happen… so, the project teams will plan their work, and come up with an end date, and the project manager will make sure that they add enough capacity to the project for the weighted risks to happen. (Q14)

Many of our interviewees made the point that 'security is not an absolute,' but that security is what the users and stakeholders need for a particular situation at a particular time. For such stakeholders to make a good decision on what security to implement requires particularly effective communication. The stakeholders will be making cost benefit trade-offs comparing various business risks.

You've got to put a weighting on the threat. You've got a level of threat, and you've got to put the appropriate level of security against that. (P4)

There are techniques available to give objective assessment of security risks, such as work by ben Othmane et al. [38]. Vitally—and several interviewees stressed this—the cost-benefit trade-offs mean that perfect security, even if possible, would rarely be a good business decision:

And actually the way this works, in practice is you have to do less than a perfect job, in order to have a measurable degree of failure or fraud or whatever, so that you can adjust your investment and say 'I am managing this to an economically viable level' because if it is zero, you have invested too much. (P6)

For simpler projects and systems, there may not be sufficient engagement from stakeholders to be able to do this kind of trade-off; in that case, it becomes the responsibility of the development team:

[Often it's impossible to get signoff on security in a big company and so the decision is usually down the developer because you can't get the signoff. And in a small company may just be the same]. Customers often don't have a view. The important thing is making the decision. (P1)

Given that each mitigation now has a cost and benefit, the decision on whether to do it becomes part of standard project management process. It is outside the scope of this technique – and indeed of the topic of software security – to explore how to make these decisions in general; the balancing of risk cost and reward is a well-understood aspect of business life.

And it has to be a bit of a trade off as well in terms of business. You've got to make the trade off as to what's good for getting a solution available now, and having one available in a year's time, which no one will buy, because everyone's gone with one which doesn't even consider security at all. (P12)

4.7 Configuration Review

Configuration Review is a review of the way a system or its software has been configured to see if this leads to known vulnerabilities, using manual checking software versions or automated build review scanners [48].

4.7.1 Context

Using an insecure component automatically makes a developed system insecure, regardless of the quality of the code developed by the team:

WordPress plug-ins are an enormous liability. Anyone can write one, most of them are rubbish. Anyone can get them put up on the plug-in directly, which gives them this air of authenticity, and quality that they don't deserve, frankly. (Q1)

So, a 'low hanging fruit' for development is to use only components (including frameworks and toolchains) that are well written and securely implemented. This is non-trivial, given the wide range of components available.

4.7.2 Solution

Configuration Review is the activity of reviewing the codebase to evaluate the components used for security: using public repositories of security vulnerabilities to assess well-known components; avoiding little-known components where possible; and using Source Code Review and Penetration Test to evaluate such components if they must be used.

4.7.3 Execution and Counterparties

For some development systems there are corresponding web sites with security reviews of plug-ins; cross referencing with these sites is a powerful security technique. This may use automated analysis tools:

[Our tool chain] also queries Wpvulndb for the plug-in that you are expecting, and tells you if there have been any published vulnerabilities in it. (Q1)

Where such sites are unavailable, or for new plug-ins, there is a value to Source Code Reviews (Section 4.8) to establish the likely security attributes of a given plug-in. This does however have a significant cost to development teams since it takes effort, however much automation may be involved, and restricts the plug-ins that that developers can use.

And so that is one of the things you end up sitting down with developers going "I'm sorry, but I know this is actually going to slow you down". And we are desperately normally trying to avoid that, I'm trying to make your lives as easy as possible. … But you have to say "well, no, you can't just add [components] – you have to review them. You don't have to do the most detailed review in the world, but if you think it looks worrying, then … don't put it in your code". (Q9)

A second issue is that, since plug-ins are widely shared, any weakness in a plug-in becomes known to attackers, and therefore it is important to keep plug-ins upgraded to the latest versions in which defects have been corrected.

We keep track of all the patches and everything for all our systems. (Q7)

Since Penetration Testing tests a whole system, it will also find vulnerabilities in components, so testing is another way to assess components:

[When Penetration Testing] you'll find the OWASP Top 10 in one [component] alone! (Q9)

4.8 Source Code Review

Source Code Review is the manual examination of source code to discover faults that may have been introduced during the software development process [48].

4.8.1 Context

It is notoriously difficult to spot one's own errors. This is especially true when the errors are faults in complex reasoning or are due to misunderstandings. Thus, a programmer working solo is likely to create avoidable security problems, just because they can naturally have only one point of view.

So, it is very easy when you are trying to deliver something yourself, as a developer, to pass over the bit that you are not doing (P5)

This problem extends to programming teams. A team will always suffer to some extent from 'groupthink;' the need to generate a shared understanding brings with it the danger that that understanding may include misunderstandings and blind spots. This is particularly important with software security, since such blind spots often lead to vulnerabilities in the developed software.

4.8.2 Solution

In Source Code Review, development teams to provide a counterpart who reviews the security and privacy aspect of assumptions, decisions, and code. Several interviewees stressed the importance of this being a dialectic, questioning, process for security reviews:

What is our most successful technique for secure software? In terms of what I have seen, certainly talking amongst the developers, the code reviews have been very useful. (Q16)

4.8.3 Execution and Counterparties

The reviewer is typically another developer from the same team.

Code review is what we do endlessly. We certainly do not let any form of code out the door, without an independent review and that is eyeballs on the code and that is discussion about the code. (P5)

If a Security Expert is available, however, they can often provide more security-relevant questioning than development team members:

The most successful technique has to be review by [a security] expert—you can't really beat that—an actual conversational review by an expert, because someone who is an expert in security might not be an expert in the domain. (P3)

[We have] 'software pen testing,' where you have some Subject Matter Experts who take a piece of code and review that in detail, and work with the developers in tandem, looking at the code and saying, "we think this is really high risk, we really want to look at this." And then somebody goes through that code with the mindset of "how do I exploit the code" (Q5)

Section 4.7 discusses Source Code Reviews of components developed externally; this might not be a detailed review, but a lightweight check, possibly backed up by forms of Vulnerability Scan (Section 4.9)

We have a tool that we wrote … for checking … plug-ins, which is intended to make code reviews of plug-ins more focused. It looks for things that are indications of badness and you go and review that list of things, rather than sitting down and reviewing the entire plug-in. (Q1)

Another approach to Source Code Review is providing dialectic questioning to a developer via pair programming, where the code is written by two developers sharing one computer:

Two heads are better than one, more eyes on the problem. (P7)

Cheaper possible alternatives are 'Rubber Duck Debugging' [25] and 'Cardboard Consultant' [53], where the developer explains their thought processes to an anthropomorphized object.

And the heavyweight approach is a formal review process, with separate review meetings delivering lists of defects for a developer to fix.

It is in the culture. We do reviews; we always have to do reviews. We set things up – and it is not regarded as a second class. (Q6)

4.9 Vulnerability Scan

Vulnerability Scan is the process of using an automated scanner on an application or network to identify vulnerabilities [48].

4.9.1 Context

It is a poor use of expensive resources to find problems that are cheap to find in other ways. Source Code Review, Configuration Review and Penetration Testing are all expensive in terms of human effort.

[There are] things that big organisations seem to not want to do, like … automated testing, code inspection – things that were always considered too expensive (P5)

Indeed experts' assessment of cost—cheap, moderate or expensive—for these three techniques, taking the modal choice in Such et al.'s survey, are that Source Code Review and Penetration Testing are expensive; Configuration Review is moderate [48].

4.9.2 Solution

Vulnerability Scan uses automated tools to look for possible security flaws in the written code. Tools to do this are sometimes called 'lint' checkers, after a UNIX tool that does extra checking for C code.

"[The most successful technique I have found is] to use various types of Lint checkers" (P7)

The experts' assessment was that Vulnerability Scan is cheap [48]. Indeed, automated tools can be used as an extension to the compilation process of the code, and many interviewees saw them as valuable in automating the removal of certain classes of security bugs.

We use excellent tooling from the Alassian stack, the Crucible Tool … We do a lot of static analysis. We review for security, we certainly do, but the point is, is that we will try to automate the removal of whole classes of problem from that. (Q3)

4.9.3 Execution and Counterparties

In this the dialectic counterparty is the tool or tool suite itself. There are now many such tools, some produced by commercial companies, supporting different languages and purposes. They work similarly to compilers, and generally use analysis techniques developed for compilers. Since codebases vary enormously in their style, requirements, and ways of using code, such tools often require significant work to configure:

We use something called Sonar which is a code inspection tool. We'd written templates and guides for our coding standards and certain patterns we are looking for in a code and we are looking for changes in the code that are greater than a certain percentage, and there are specific bits of the code we are looking for any change that should never happen. (P5)

Such tools typically generate large numbers of 'false positive' warnings. So, developers need to use them as questions ('is this piece of code OK') rather than as notifications of changes to be made. Hence in this paper we regard them as a dialectic counterparty, rather than merely an error detector.

Pay attention to the warnings, pay attention to the Link errors. [So it is not just the automated checks. It is the attitude towards those automated checks, taking them really seriously] Use them, don't forget them. (P7)

Some interviewees, however, had found the time cost of analysing the responses to be too great:

If you mean static analysis type tools, the answer is no [we do not use them]. Two teams I have worked with have used them. One gave up entirely, because they looked at the amount of time that they were spending on it, versus the reward that they were getting, and then looked at what happened in the first pen test that they had, where they just got completely pwned. (Q9)

4.10 Penetration Testing

A Penetration Test is a simulated attack on a component or system, carried out by a security expert using similar techniques to that of a real-world malicious attacker [48].

4.10.1 Context

All the assurance techniques discussed so far (Sections 4.5 to 4.9) need access to the development team or source code. Yet the delivered system is what the attackers see and where the privacy issues occur.

I teach … the application security or hackers' mindset: 'okay, I have got this application in front of me. How would I go about finding flaws, what are the assumptions they are making, how do we go about testing the assumptions?' (P11)

So, there are situations where none of the other assurance techniques will help.

4.10.2 Solution

In Penetration Testing an external 'white hat' security team simulates what an attacker would do to attempt to gain access or disable the service. The team then feeds back any 'successful' exploits they have found to the development and operations teams.

[Ensuring software security] tends to get handed off, in most companies I've worked with, to a white-hat hacking team. (P7)

[The most successful intervention technique we have found] comes down to using security experts. We … have Penetration tests. (Q5)

If the team have developed something new… and it is a significant change, we might get it externally pen tested, if we think that we can't test it ourselves. (Q12)

4.10.3 Execution and Counterparties

The single dialectic counterparty is a Penetration Test Expert. Few developers have more than basic skill at Penetration Testing, so it would be unusual for a development team member to take the role. The developers take the results of the Penetration Test and use them to plan future security enhancements.

Normally the way the report would be dealt with, we would sit down in a sprint planning meeting and we would go through the report: us, the client, sometimes the tester, … and we would discuss the findings and produce work that would be added to the next sprint to address what had been found. (Q1)

At the operating system level, one can also Penetration Test a mobile operating system using software on the device:

I think the one [approach to testing our phones] that has been, arguably, most useful has been using specialist external consultancy around security. Not for training, but "can you just come in and penetration test this device?" (P2)

Some interviewees have success extending the Penetration Testing to involve direct discussion (dialectic) with the developers:

So, the idea is the penetration tester is testing the web application, and the developer is sitting with the penetration tester. That is very effective. A developer will not surely not know more about the application and will see – I am not such a good pen tester, but I do not claim to be able to find all the vulnerabilities in the application – there may be something though that is too arcane for me in 5 days or 3 days to understand from a black box perspectives, but if you are sitting with a developer they are part of the process and they tell you 'ah, you know that request you have just intercepted in Burp - I'll change it to that!' (Q11)

4.11 Summary of Assurance Techniques and Dialectic Counterparties

Table 2 summarises the Dialectical Security theory, showing which counterparties provide the dialectic questioning for each of the assurance techniques discussed in Sections 4.5 to 4.10.

Table 2: Dialectic Security Assurance Techniques and their Counterparties

	Members of Development Team	Stakeholders, e.g. Product Managers	Automated Code Analysis Tools	Security Experts	Penetration Test Experts	Other Dev. Teams
Threat Assessment	■	■		■	■	
Stakeholder Negotiation*	■	■		■		
Configuration Review	■			■		
Source Code Review	■			■		■
Vulnerability Scan			■			
Penetration Test					■	

Note that end users and operations staff were not mentioned in the context of any of the Assurance Techniques; they are omitted from the table.

5 Discussion

5.1 Differences between the Two Surveys

Section 4.3 shows remarkable differences between the two surveys. Even P6/Q2 (one person) discussed Vulnerability Scan in the second interview and not in the first. We suggest these are attributable mainly to the different questions asked (see Appendix). It appears that when experts review developer security in general, they tend to focus on Threat Assessment and Configuration Review; when they come to consider aspects of development security in more detail they then focus more on other techniques, especially Vulnerability Scan.

It is interesting that many of the second survey's participants emphasised Stakeholder Negotiation, even though the nearest that survey's interview protocol came to mentioning it was 'Security coordination with other teams' (where the survey designers had in mind security experts and other developers). That, and the lack of emphasis on it in the first survey, suggest that Stakeholder Negotiation is an important 'emergent requirement' for software development security when practitioners consider the practical detail of implementing interventions.

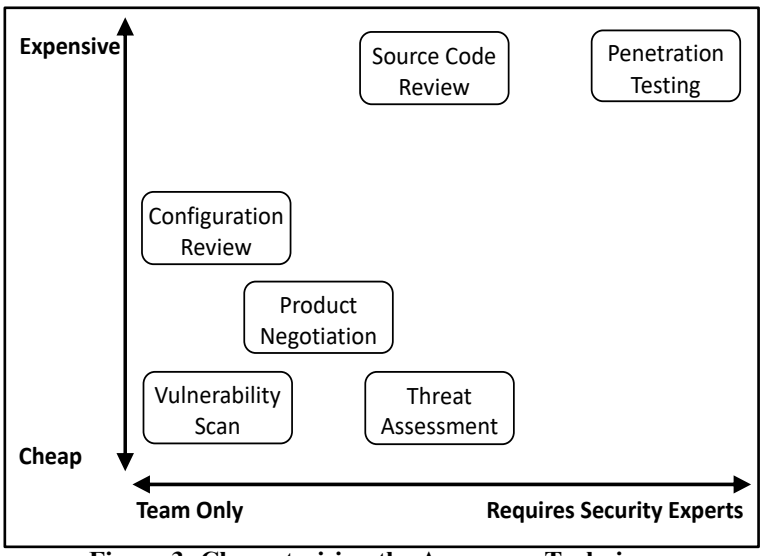

Figure 3: Characterising the Assurance Techniques

5.2 Choosing Assurance Techniques

We observe that the list of key assurance techniques derived in Section 4.2 is nevertheless a small subset of the 20 techniques identified by Such et al. [48]: the explicit focus that experts placed on them is therefore important.

Based on Table 2 and the expert cost estimates from Such et al., we can further characterise the assurance techniques in terms of two important practical considerations: their cost and their need for security expertise.

Figure 3 shows this characterisation, using the modal estimate of cost for each technique, and assigning 'cheap' to Product Negotiation, since it in incorporated into activities already carried out by a development team. Source Code Review, Threat Assessment and to a lesser extent Product Negotiation, are more effective with the involvement of security experts, but can take place without them; Penetration Testing requires security expertise, whether in-house or outsourced.

Based on this, we deduce that the most widely applicable assurance techniques are Threat Assessment, Product Negotiation, Configuration Review and Vulnerability Scan; and that teams with greater resources will benefit from adding Source Code Review and Penetration Testing.

5.3 Incorporating the Assurance Techniques into the Development Cycle

The ordering of the assurance techniques in Section 4.2 is roughly chronological within a development cycle. Figure 4 illustrates how they might be incorporated into an iterative cycle. Threat Assessment and Product Negotiation affect what functionality is produced, so apply to the planning element; Configuration Review and Vulnerability Scan can be automated into a product build; Source Code Review needs a candidate complete implementation, so is typically done at the release stage; and Penetration Testing applies to an installed system, so comes in the test phase.

Section 4.11 discusses the dialectic counterparties involved in each assurance technique, and observes that end users and operations staff were not mentioned. We speculate that this reflects the difficulty of involving them, or possibly that it represents an opportunity not yet realised by the security specialists we interviewed.

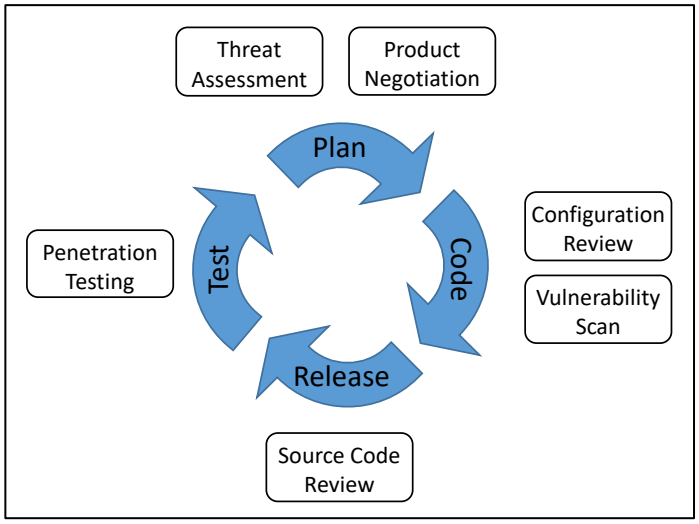

Figure 4: Assurance Techniques within the Development Cycle

5.4 Improvements on Existing Practice

Considering the assurance techniques, past improvements to developer based security have often been based purely on the intervener proving the inadequacy of existing security using Penetration Testing, an approach that often proved ineffective in the longer term (see Section 2.3). Dialectical Security suggests an alternative way of approaching security improvement: regarding it as an interactive dialog with the developers, where all the techniques, from Threat Assessment through to Penetration Testing, are regarded as generating challenges for the developers to debate, discuss and address. The surveys described in this article demonstrate that expert opinion considers this approach to be more successful.

5.5 Relationship to Existing Work

Comparing the existing work on developers and security, Section 2.1 discussed research on current practice by developers such as that by Balebako et al. [6]; this work goes further by identifying approaches for *better* practice.

Considering the assurance techniques themselves, we suggest that all but Stakeholder Negotiation are reasonably well understood [48]. There has been, however, little work on the human interactions required to achieve the techniques, a gap filled by this article.

The work of Such et al. [48] provides a taxonomy and cost basis for assurance techniques and identifies the set of Architectural Review, Configuration Review, Penetration Testing and Vulnerability Scan as the most cost effective. By deeper and more intensive questioning of the subjects, this article reaches a similar but slightly different conclusion, that the subset Threat Assessment, Product Negotiation, Configuration Review, Vulnerability Scan, Source Code Review and Penetration Testing is likely to be the most effective.

As discussed in Section 2.3, there has been a good deal of work on ‘Secure Software Development Lifecycle’, but prescriptive SSDLs have not been successful with development teams. This work identifies a set of assurance techniques that has been successful. It also, by doing so, provides a basis for future work finding the most effective ways to introduce those techniques to teams that are not yet using them.

The work by Ashenden and Lawrence [5] provides valuable suggestions how to improve the effectiveness of the relationships between security professionals and software developers; Dialectical Security goes further by proposing ways to leverage such improved communication to improve the deliverables from software developers.

5.6 Possible Future Work on Dialectic

The importance of dialectic in secure development justifies further research into ways best to use human and team interactions to improve security. Future research might ask research questions such as:

- What quantitative evidence can we find that dialectic techniques lead to better security outcomes?
- What are the most effective ways to generate understanding of attackers and potential exploits?
- How best do we represent security questions in business terms?
- What forms of cross-team interaction are most effective to ensure app security?

There are several possible approaches to this research. An ethnographic approach might follow the progress of a development team, identifying where the major security mitigations were identified and how the negotiations took place in practice. Alternatively, a survey approach might ask the questions of a variety of developers and stakeholders to produce a possible consensus.

A different and potentially rewarding area of research is in further dialectic techniques for software security. For example, De Bono [11] has defined a variety of 'Lateral Thinking' techniques to help teams and individuals to challenge their thinking; these might be effective in Threat Assessment. Such research might also investigate techniques to use other counterparties such as operations staff and end users, which are apparently not widely used currently (see Section 4.11).

6 Threats to Validity

How certain can we be that this analysis accurately reflects reality? We approach this question by analysing threats to validity.

Considering first Conclusion Validity, do the research data justify the conclusions? Grounded Theory's rigorous process of line-by-line coding, categorization, and core category generation leads to a theory that does reflect the interview data. The use of extensive quotations ensures that this can be at least partially checked.

In terms of Construct Validity, do the Dialectical Security theory and the descriptions of assurance techniques represent actual practice? Grounded Theory handles this primarily in terms of 'theoretical saturation,' reached when new interviews do not add substantially to the theory. Guest [23] suggests that a dozen interviews are often sufficient for this; in this case as researchers we believe we have reached theoretical saturation with regard to the list of techniques, but not with regard to all the potential detail to be uncovered within each technique.

In terms of External Validity, can the results be generalized to a wider scope? Grounded Theory's conclusions are always limited to the specific scope studied [12]. We therefore selected interviewees from a wide range of industry roles (see Figure 1 and Table 1), via a range of mechanisms (see Figure 2), and used completely open questions (see Appendix). We conclude that the results are likely to be generalisable to development teams similar to those we interviewed, specifically to teams in the UK, Germany and the USA.

7 Conclusion

From a rigorous study using interviews of experts in secure app development, this article derives a theory, Dialectical Security, to provide a basis for analysis of developer assurance techniques. Specifically, it concludes that:

- Developer security depends on dialectic: friendly challenges to developer thinking by a variety of counterparties; and
- Six assurance techniques, a small subset of the many in use in industry, are sufficient to provide this dialectic and to deliver software development security.

This is important for modern software development ecosystems in two ways. Identifying the need for dialectic means that addressing security can move from being about formal processes, artefacts and reports, to being about the developers' interactions with counterparties (including each other) and how best to leverage those interactions. And the identification of a small number of preferred assurance techniques makes it realistic for teams to introduce these techniques with a minimum of support from security experts.

Based on the insights in these surveys, then, what needs to be done is to find practical ways to help development teams to adopt these assurance techniques, and to incorporate the techniques' dialectic into normal development. Initially this can be through direct training or consultancy-style involvement between researchers and development teams, such as that described in Weir et al. [52]. Later, to gain a wider impact, there are approaches we have previously proposed [56] to encourage adoption by a large population of developers, such as educational online games for developers to play; storytelling, through blogs or even TV; and massively online courses or TED-style video.

Focussing on Dialectical Security's six assurance techniques and the interaction they promote will, we believe, enhance the security of software as it is developed, and lead to better privacy and safety for all of us.

References

[1] Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M.L., and Stransky, C. You Get Where You're Looking For: The Impact of Information Sources on Code Security. *IEEE Symposium on Security and Privacy*, (2016), 289–305.

[2] Acar, Y., Stransky, C., Wermke, D., Weir, C., Mazurek, M.L., and Fahl, S. Developers Need Support, Too: A Survey of Security Advice for Software Developers. *IEEE Secure Development Conference*, (2017), 22–26.

[3] Anderson, R. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2008.

[4] Apple. Introduction to Secure Coding Guide. https://developer.apple.com/library/mac/documentation/Security/Conceptual/SecureCodingGuide/Introduction.html.

[5] Ashenden, D. and Lawrence, D. Security Dialogues: Building Better Relationships between Security and Business. *IEEE Security & Privacy 14*, 3 (2016), 82–87.

[6] Balebako, R., Marsh, A., Lin, J., Hong, J., and Cranor, L. The Privacy and Security Behaviors of Smartphone App Developers. *Internet Society*, October (2014).

[7] Banks, A. and Edge, C.S. *Learning iOS Security*. Packt Publishing, Birmingham, UK, 2015.

[8] Barua, A., Thomas, S.W., and Hassan, A.E. *What Are Developers Talking About? An Analysis of Topics and Trends in Stack Overflow*. 2012.

[9] Beecham, S., Baddoo, N., and Hall, T. Motivation in Software Engineering: A Systematic Literature Review. *Information and Software Technology 50*, 9 (2008), 860–878.

[10] Bell, L., Brunton-Spall, M., Smith, R., and Bird, J. *Agile Application Security: Enabling Security in a Continuous Delivery Pipeline*. O'Reilly, Sebastopol, CA, 2017.

[11] De Bono, E. *Lateral Thinking: Creativity Step by Step*. Harper & Row, 1970.

[12] Charmaz, K. *Constructing Grounded Theory*. Sage, London, 2014.

[13] Conradi, R. and Dybå, T. An Empirical Study on the Utility of Formal Routines to Transfer Knowledge and Experience. *ACM SIGSOFT Software Engineering Notes 26*, 5 (2001), 268–276.

[14] Cooperrider, D.L. and Whitney, D. *Appreciative Inquiry: A Positive Revolution in Change*. ReadHowYouWant, 2005.

[15] Do, L.N.Q., Ali, K., Livshits, B., Bodden, E., Smith, J., and Murphy-Hill, E. Just-in-time Static Analysis. *26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, (2017), 307–317.

[16] Drake, J.J., Lanier, Z., Mulliner, C., Fora, P.O., Ridley, S.A., and Wicherski, G. *Android Hacker's Handbook*. John Wiley & Sons, Indianapolis, 2014.

[17] Enes, P. and Conradi, R. Acquiring and Sharing Expert Knowledge. 2005. http://www.idi.ntnu.no/grupper/su/fordypningsprosjekt-2005/aanes-fordyp05.pdf.

[18] Forbes. Top 2016 Cybersecurity Reports Out From AT&T, Cisco, Dell, Google, IBM, McAfee, Symantec And Verizon. *Forbes*, 2017. https://www.forbes.com/sites/stevemorgan/2016/05/09/top-2016-cybersecurity-reports-out-from-att-cisco-dell-google-ibm-mcafee-symantec-and-verizon/.

[19] Geer, D. Are Companies Actually Using Secure Development Life Cycles? *IEEE Computer June*, 2010, 12–16.

[20] Glaser, B.G. and Strauss, A.L. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Transaction, Chicago, 1973.

[21] Gollmann, D. *Computer Security*. Chichester : Wiley, 2011.

[22] Google. Android Security Tips. http://developer.android.com/training/articles/security-tips.html.

[23] Guest, G., Bunce, A., and Johnson, L. How Many Interviews are Enough? An Experiment with Data Saturation and Variability. *Field methods 18*, 1 (2006), 59–82.

[24] Howard, M., LeBlanc, D., and Viega, J. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill, Inc., 2009.

[25] Hunt, A. and Thomas, D. *The Pragmatic Programmer : From Journeyman to Master*. Addison-Wesley, 2000.

[26] ISO/IEC. ISO/IEC 21827:2008 - Systems Security Engineering - Capability Maturity Model. *2008*, (2008), 144.

[27] Johnson, M. and Senges, M. Learning to Be a Programmer in a Complex Organization. *Journal of Workplace Learning 22*, 3 (2010), 180–194.

[28] Lerch, J., Hermann, B., Bodden, E., and Mezini, M. FlowTwist: Efficient Context-sensitive Inside-out Taint Analysis for Large Codebases. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (2014), 98–108.

[29] Lopez, T., Sharp, H., Tun, T., Bandara, A., Levine, M., and Nuseibeh, B. Talking about Security with Professional Developers. *7th International Workshop Series on Conducting Empirical Studies in Industry (CESSER-IP)*, IEEE Computer Society (2019).

[30] McGraw, G. Four Software Security Findings. *Computer 49*, 1 (2016), 84–87.

[31] McGraw, G., Migues, S., and West, J. *Building Security In Maturity Model (BSIMM7)*. 2016.

[32] Microsoft. Learning Security - MSDN. https://msdn.microsoft.com/en-us/security/aa570420.aspx.

[33] Microsoft. Microsoft Secure Development Lifecycle. https://www.microsoft.com/en-us/sdl/.

[34] Microsoft. *Microsoft Security Intelligence Report, Volume 23*. Redmond, WA, 2018.

[35] Murphy-Hill, E., Lee, D.Y., Murphy, G.C., and McGrenere, J. How Do Users Discover New Tools in Software Development and Beyond? *Computer Supported Cooperative Work (CSCW) 24*, 5 (2015), 389–422.

[36] Near, J.P. and Jackson, D. Finding Security Bugs in Web Applications Using a Catalog of Access Control Patterns. *Proceedings of the 38th International Conference on Software Engineering*, ACM (2016), 947–958.

[37] Nguyen, D.C., Wermke, D., Backes, M., Weir, C., and Fahl, S. A Stitch in Time: Supporting Android Developers in Writing Secure Code. *ACM SIGSAC Conference on Computer & Communications Security*, ACM (2017).

[38] Ben Othmane, L., Ranchal, R., Fernando, R., Bhargava, B., and Bodden, E. Incorporating Attacker Capabilities in Risk Estimation and Mitigation. *Computers & Security 51*, (2015), 41–61.

[39] OWASP. Mobile Security Project - Top Ten Mobile Risks. https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks.

[40] OWASP. Software Assurance Maturity Model Project. https://www.owasp.org/index.php/OWASP_SAMM_Project.

[41] Poller, A., Kocksch, L., Türpe, S., Epp, F.A., and Kinder-Kurlanda, K. Can Security Become a Routine? A Study of Organizational Change in an Agile Software Development Group. *ACM Conference on Computer Supported Cooperative Work*, (2017), 2489–2503.

[42] Ponemon Institute. *The State of Mobile Application Insecurity*. 2015.

[43] Pribik, I. and Felfernig, A. Towards Persuasive Technology for Software Development Environments: An Empirical Study. *International Conference on Persuasive Technology*, Springer (2012), 227–238.

[44] Schneier, B. *Secrets and Lies: Digital Security in a Networked World*. John Wiley & Sons, 2011.

[45] Smeets, Y.R. Improving the Adoption of Dynamic Web Security Vulnerability Scanners. 2015.

[46] Steel, C., Nagappan, R., and Lai, R. *Core Security Patterns*. Prentice Hall, 2006.

[47] Stol, K., Ralph, P., and Fitzgerald, B. Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. *38th International Conference on Software Engineering*, ACM (2015), 120–131.

[48] Such, J.M., Gouglidis, A., Knowles, W., Misra, G., and Rashid, A. Information Assurance Techniques: Perceived Cost Effectiveness. *Computers and Security 60*, (2016), 117–133.

[49] Türpe, S., Kocksch, L., and Poller, A. Penetration Tests a Turning Point in Security Practices? Organizational Challenges and Implications in a Software Development Team. *Second Workshop on Security Information Workers, Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, USENIX Association (2016).

[50] Veracode. *State of Software Security Report Volume 9*. 2018.

[51] Votipka, D., Stevens, R., Redmiles, E., Hu, J., and Mazurek, M. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE (2018), 374–391.

[52] Weir, C., Blair, L., Becker, I., Sasse, M.A., and Noble, J. Light-touch Interventions to Improve Software Development Security. *IEEE Cybersecurity Development Conference*, IEEE Computer Society (2018), 12.

[53] Weir, C. and Noble, J. Process Patterns for Personal Practice. *Proceedings of the 4th European Conference on Pattern Languages of Programming*, UVK - Universitaetsverlag Konstanz (1999), 413–424.

[54] Weir, C., Rashid, A., and Noble, J. How to Improve the Security Skills of Mobile App Developers: Comparing and Contrasting Expert Views. *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016): Workshop on Security Information Workers*, USENIX Association (2016).

[55] Weir, C., Rashid, A., and Noble, J. Early Report: How to Improve Programmers' Expertise at App Security? *1st International Workshop on Innovations in Mobile Privacy and Security co-located with the International Symposium on Engineering Secure Software and Systems (ESSoS 2016)*, CEUR-WS.org (2016), 49–50.

[56] Weir, C., Rashid, A., and Noble, J. Reaching the Masses: A New Subdiscipline of App Programmer Education. *FSE'16: 24nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering Proceedings: Visions and Reflections*, ACM (2016).

[57] Wikipedia. Dialectic. https://en.wikipedia.org/wiki/Dialectic.

[58] Xie, J., Chu, B., Lipford, H.R., and Melton, J.T. ASIDE: IDE Support for Web Application Security. *27th Annual Computer Security Applications Conference*, (2011), 267.

[59] Xie, J., Lipford, H.R., and Chu, B. Why Do Programmers Make Security Errors? *IEEE Symposium on Visual Languages and Human Centric Computing*, (2011), 161–164.

[60] Xie, J., Lipford, H.R., and Chu, B.B.-T. Evaluating Interactive Support for Secure Programming. *SIGCHI Conference on Human Factors in Computing Systems*, ACM (2012), 2707–2716.

[61] Yskout, K., Scandariato, R., and Joosen, W. Do Security Patterns Really Help Designers? *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, (2015), 292–302.

Appendix: Interview Protocols

First Phase Protocol: App Security Expert Interview Questions

Introduction – establish context

- Please could you tell me something of your own background?
- What is your current role, and what do you find yourself doing day-to-day?
- How did you first get involved with developing secure software?

Exploration

- Can you think of a particular triumph in your work? How did you achieve the security aspects of that?
- Please could you give examples of a secure system that has gone well? Or not gone well and been fixed?
- How did you initially learn about adding security to software development?
- How do you learn more now?
- What aspects of your team (your work) made them particularly good at secure software?
- What is the most successful technique you have found?
- What advantages are there in the development of the app end (browser or mobile) over the server end?

Clarification

- You mentioned [specific technique]. Can you tell me a little more about that?

Second Phase Protocol: Intervention Expert Interview

Introduction – establish context

- What is your current role, and what do you find yourself doing day-to-day? Tell me about a typical day at work?
- Briefly, how did you first get involved with secure software development?

Exploration

- What's your interest in security? What do you do about it, and how do you deal with it day-to-day?
- What do you want to achieve when you're helping a team improve software security? How do you define and measure success?
- What is the most successful intervention technique you've found? Where do you concentrate your efforts?
- Can you think of a particular triumph in your work – where you've worked with a team that has improved their security? How did you achieve that?
- Have any of your teams used code checking tools? How happy were you with their effectiveness at finding problems; and their ease of use?
- What do you find effective as motivation for secure development?
- How do you frighten developers into security, or emphasise the positive aspects?
- To what extent are laws and standards helpful in getting teams to be effective at software security? How do you find out about them and keep up to date?
- When new people join an existing team, how do you motivate them and how do they learn what's required? Do you encourage double checking of contributions from new people or treat them "as usual"?
- What are the best ways you've found to get teams to tackle specific things:
  - Security coordination with other teams;
  - Reviews and penetration testing;
  - Designing to get feedback from the users?
  - What else?
- Have you had a nightmare scenario? Or consider this nightmare scenario. You're working with a team that's just learned they have a security flaw in a website that's very heavily used. Have you even had a situation like that (no details required)? What did or would you do to help the team tackle it?

Vision

Let's imaging we're a few years in the future, and the problem of getting teams up to speed with app security has been licked; it's now a part of everyday software development life.

- How was it done?
- What were the first small steps?

Clarification (as appropriate)

- And how did you achieve that?
- Oh I see. Could you give an example?